Designing Your Application for Concurrency

Session 425

These are confidential sessions—please refrain from streaming, blogging, or taking pictures.

Designing Your Application for Concurrency

Ben Trumbull
Engineering Manager, Core Data

Why Concurrency?

- Performance
  - Responsiveness
  - Speed
  - Efficiency
- “Spend” time saved
  - New features
  - Better user experience

Subdividing Your Application

- Pieces too big
  - Less concurrency
- Pieces too small
  - More overhead
  - More complexity

Finding the Right Granularity

Task Granularity

- Limited by overhead
- Coordination mechanisms
- Resource contention

Road Map

- Cast of characters
- Coordination mechanisms
- Resource contention
- Task granularity
- Notifications
- UI layer

Cast of Characters

Basic Introductions

- Tasks
- Threads
- Dispatch queues
- Operation queues
- Locks
- Atomic operations
- Futures

Tasks

- Work unit
- Distinct grouping of code
- Executed by a thread or queue
- Today, not an NSTask

Threads

- Thread of execution
- Independent flow of control
- Shared address space

Dispatch Queues

- Grand Central Dispatch
- New in Snow Leopard
- Easier, more efficient than threads
- Library for common operations

Operation Queues

- NSOperationQueue
- Cocoa API for work queues
- Uses Grand Central Dispatch on Snow Leopard
- Additional functionality

Locks

- Simple protection by exclusion
- Usually only permit one thread at a time
- Many types available

Atomic Operations

- Hardware supported primitives
  - Basic arithmetic operations
  - Compare and swap
- Each address acts like encapsulated lock

Futures

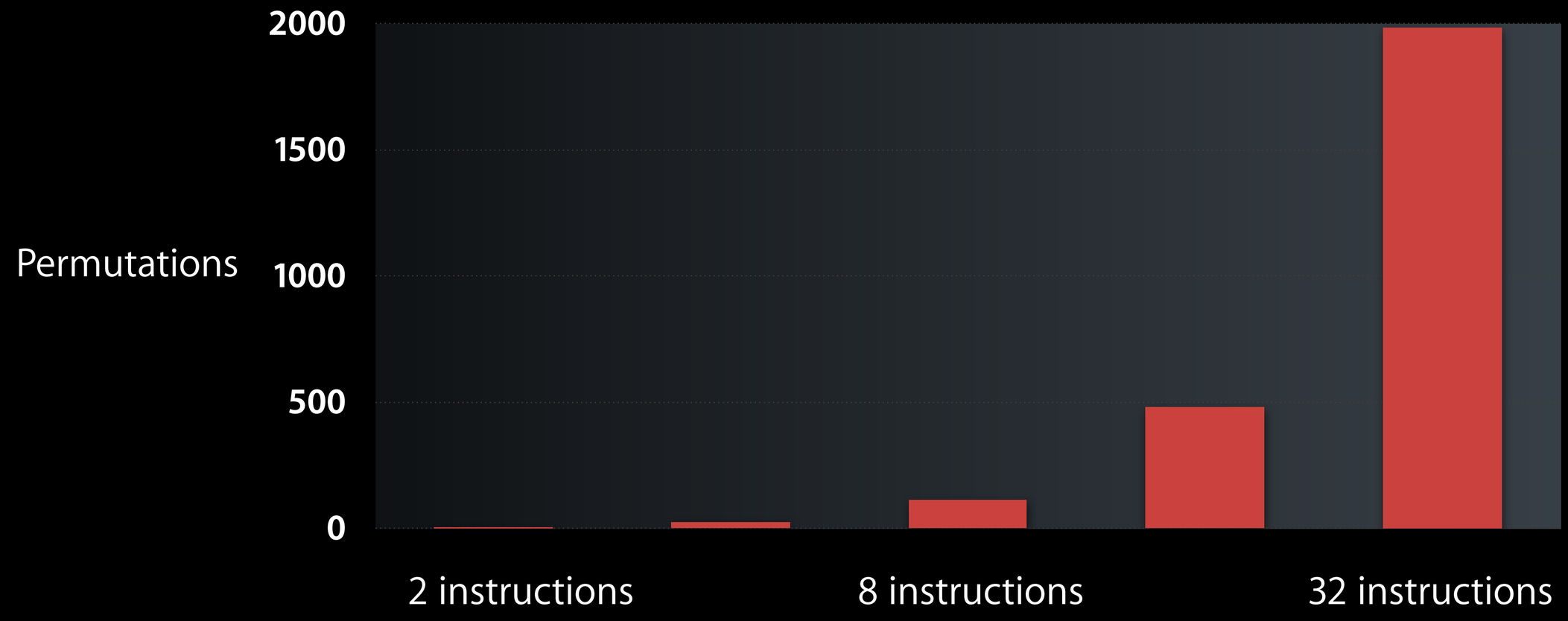
- Placeholder for unknown value
- Often return value for asynchronous APIs
- Improves responsiveness
- No blocking until absolutely required

Coordination Mechanisms

Problems with Threads

- Threads are non-deterministic
- Execution interleaved randomly
- Sharing resources and data has many side effects

Two Threads Interleaving



Coordination Goals

- Prune complexity
- Reduce non-determinism
- Safely share resources

Locking

Locking

- Prevents simultaneous access
- Most common type of coordination
- Multiple independent locks prone to deadlock

Some Common Locks

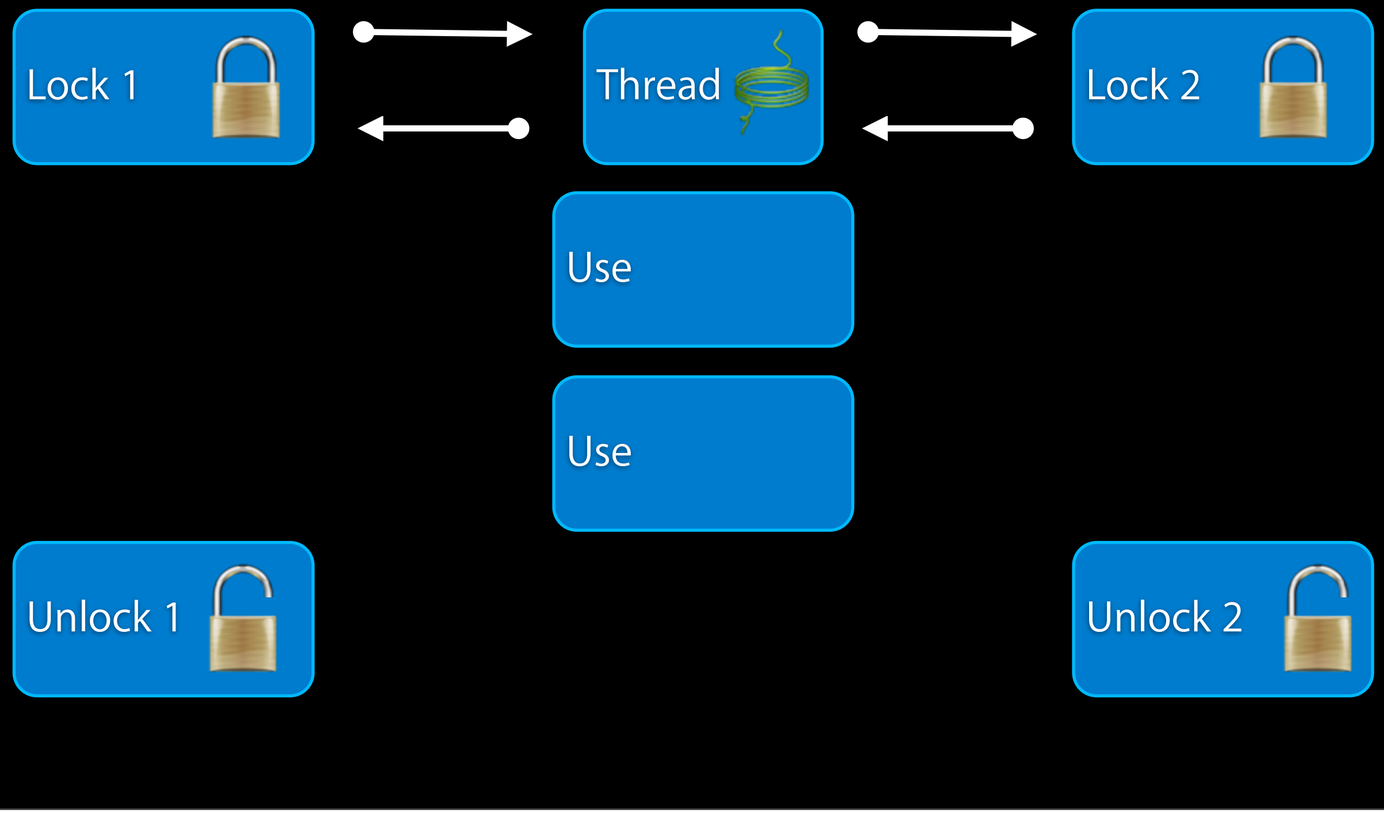
- @synchronized
- NSLock
- OSSpinLock
- pthread_rwlock

Documentation

Mac OS X “Threading Programming Guide”

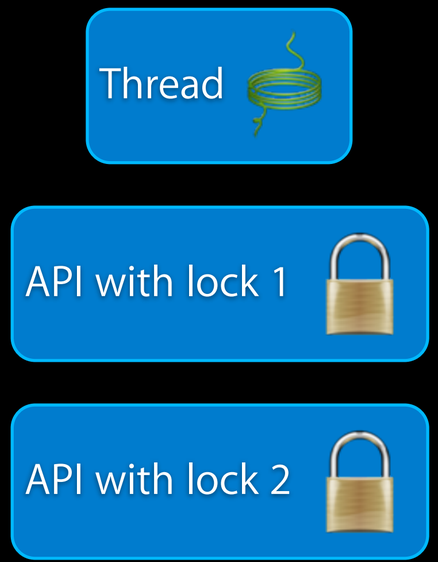
Independent Locks

- Multiple locks acquired and released in any order
- Allows calling other code within scope



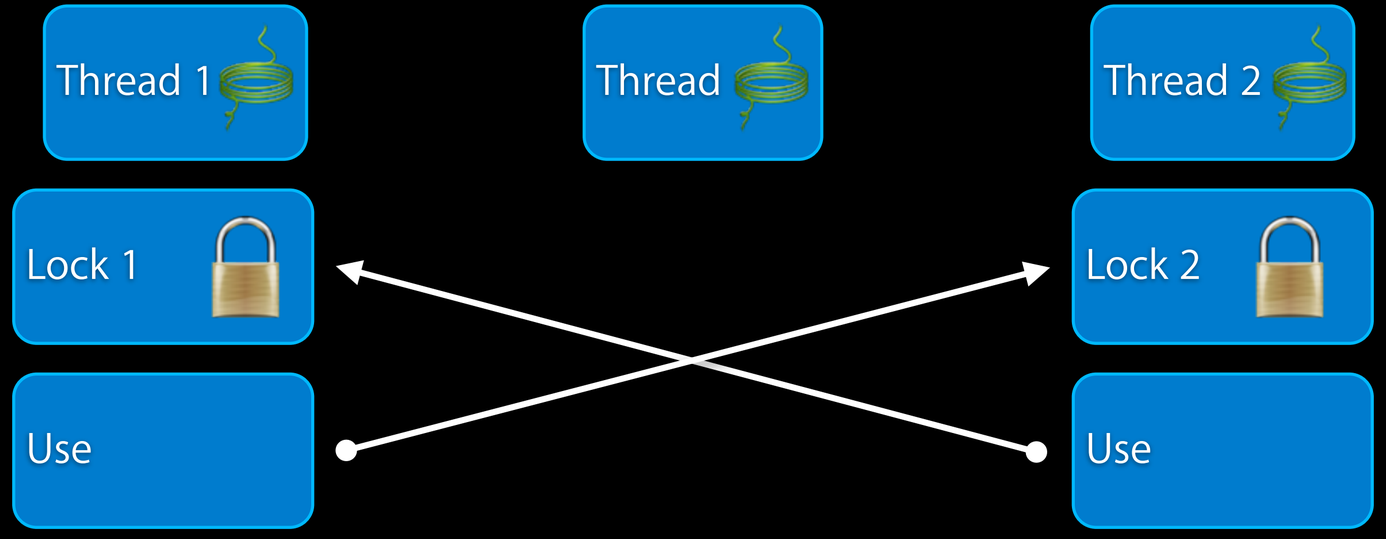
Encapsulated Locks

- Hidden behind API
- Released at end of scope
- No call outs



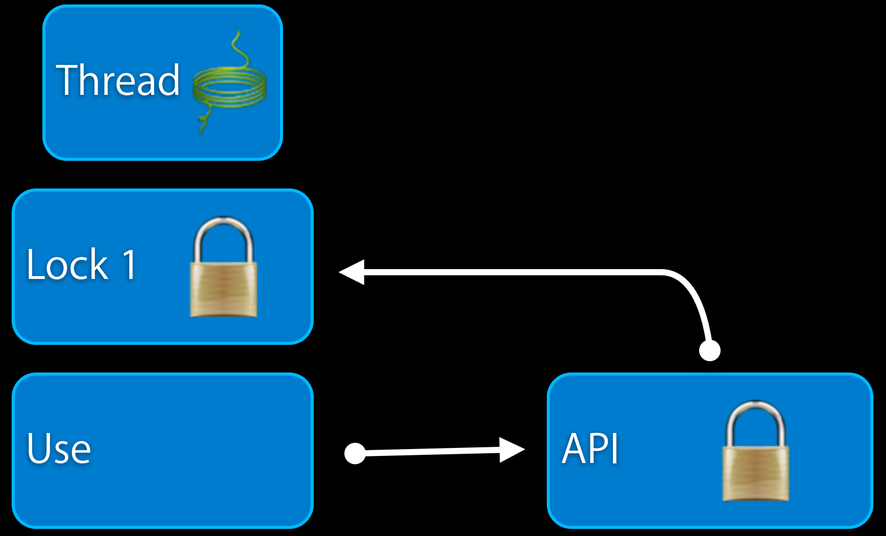
Deadlock

- Two threads each have half the necessary locks



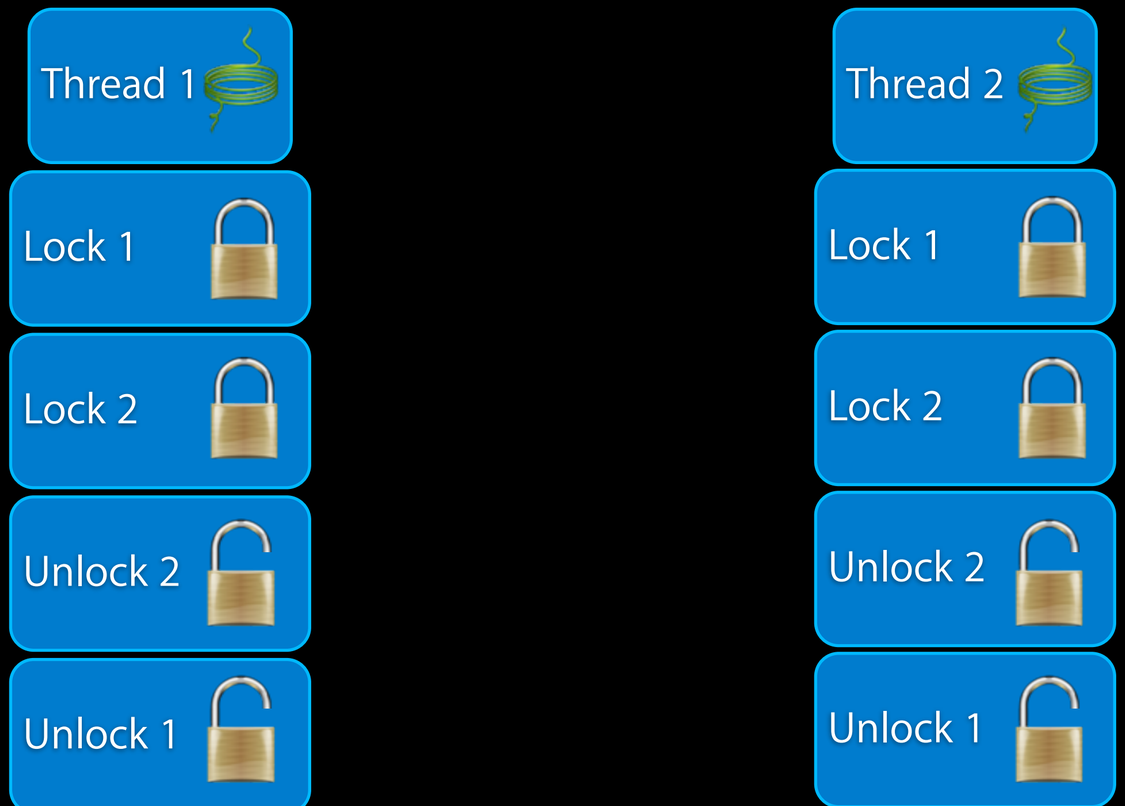
Single Thread Deadlock

- Non-recursive independent locks
- Common with spin locks



Lock Ordering

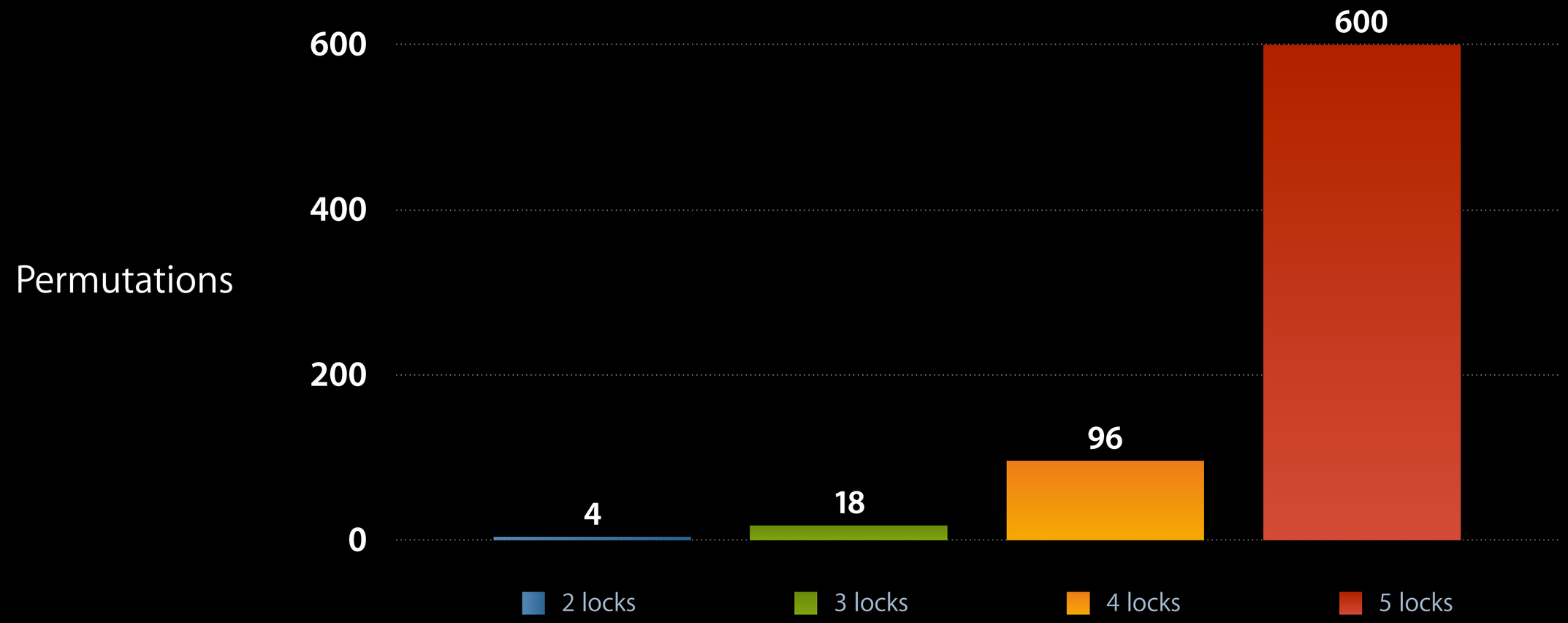
- Inconsistent lock ordering for different code paths deadlocks
- Acquire in fixed order
- Release in inverse order



Limits of Lock Ordering

- Ordering protocol must be consistent
- Fragile with maintenance and new features
- Avoid more than two independent locks

Unconstrained Lock Permutations



Complexity Increases Exponentially

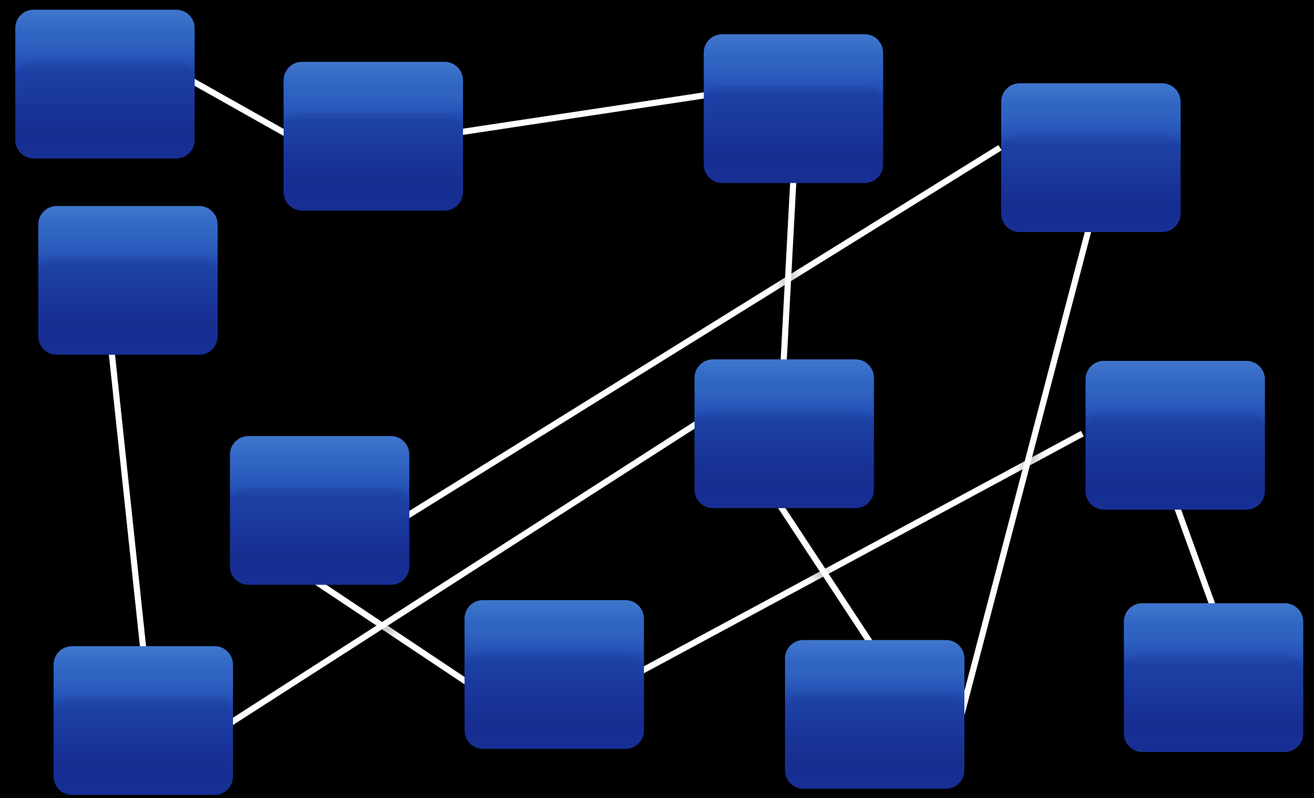
- Three independent locks have 18 permutations
- One-third works correctly

Locks

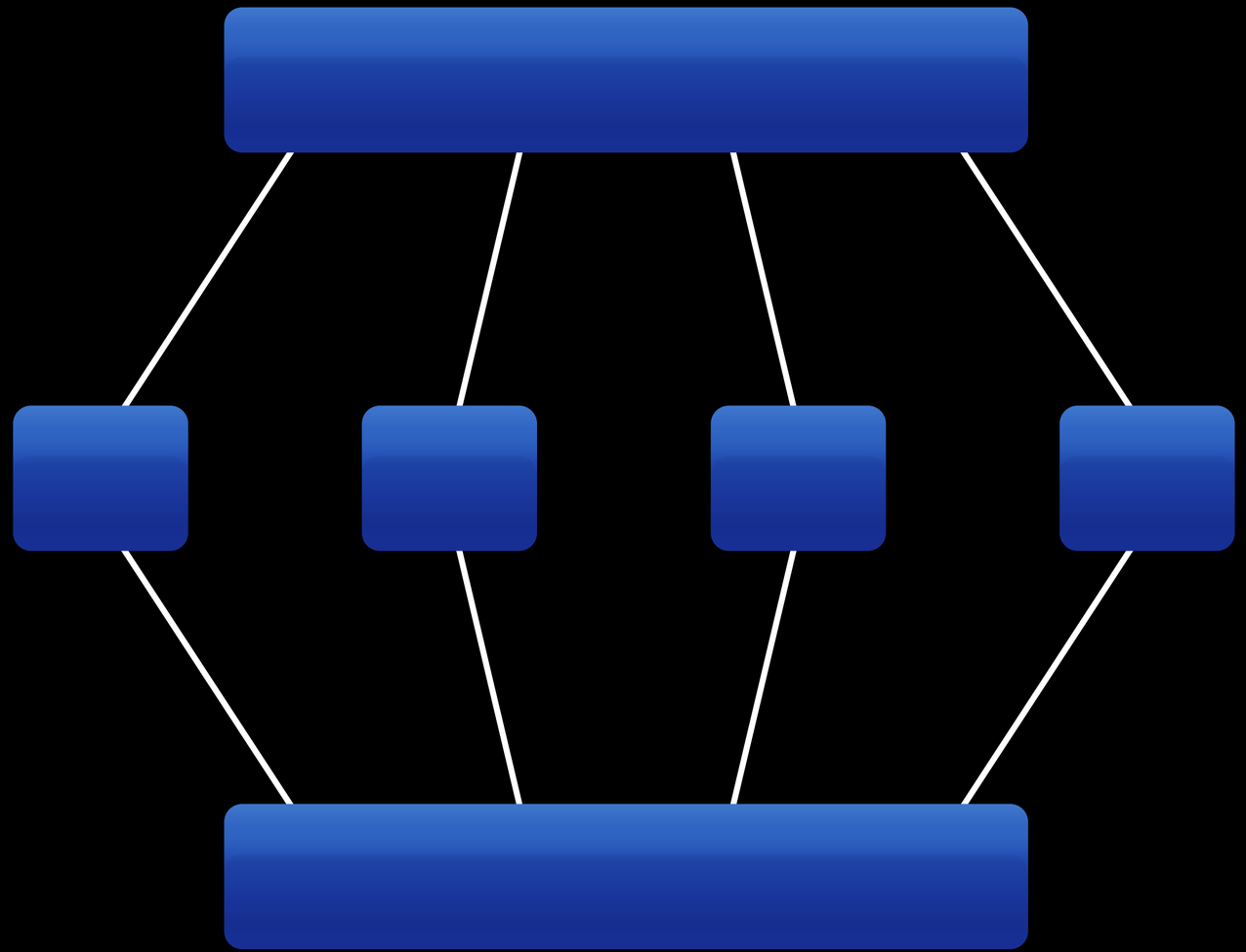
- Many types available
- Prefer encapsulated locks
- Order independent locks

Isolation

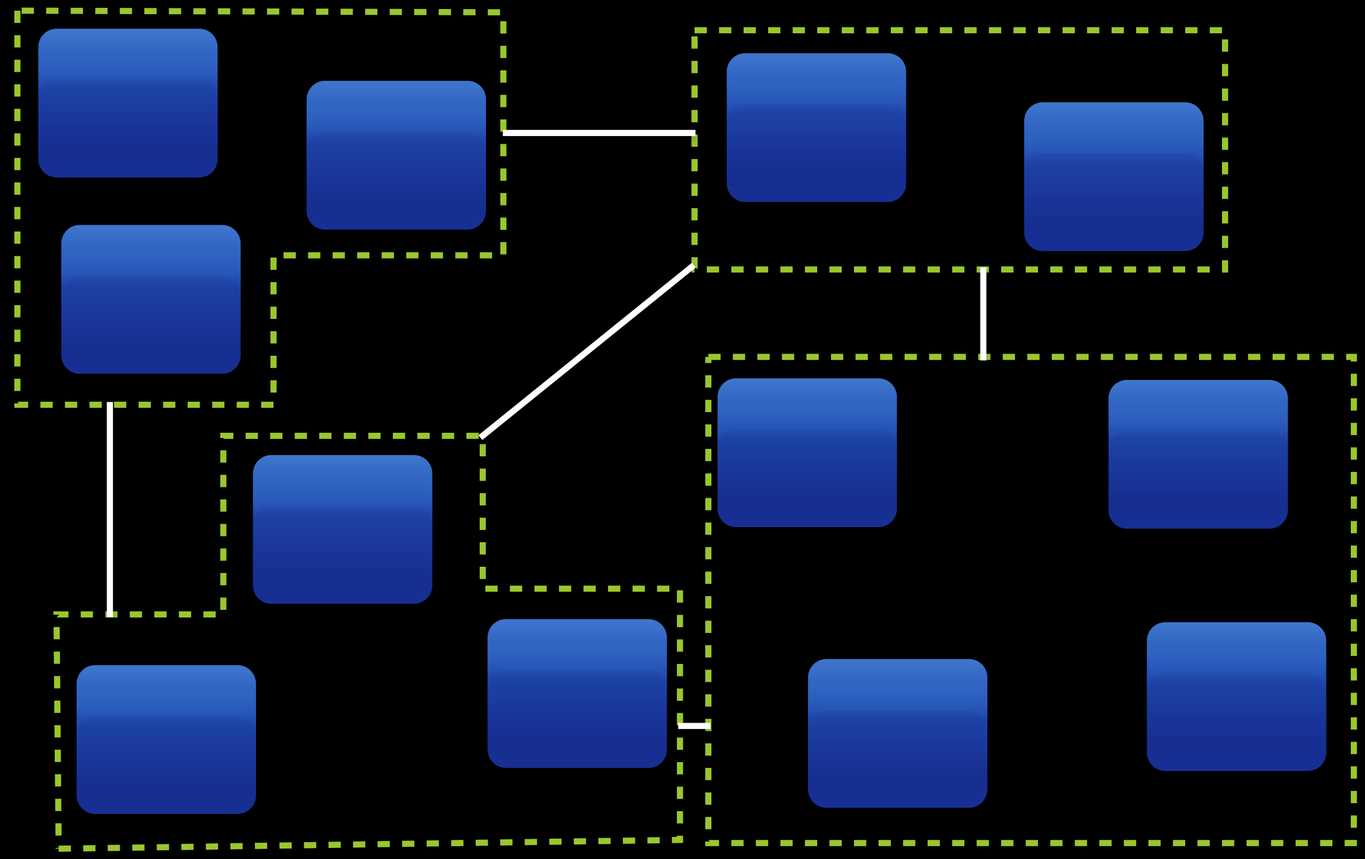
Turn Complex Interactions



Into Something Simple



Or at Least Manageable



Isolation

- Subdivide multithreaded app into serial pieces
- Reduces complexity
- Isolation is to concurrency what encapsulation is to OOP

Isolation (Confinement)

- Confine all access to resource to one thread
- No arbitrary shared memory access
- Communication only by formal request

Alternatives to Sharing

- Copy mutable data
- Pair tasks with an owned resource
  - Resource centric view

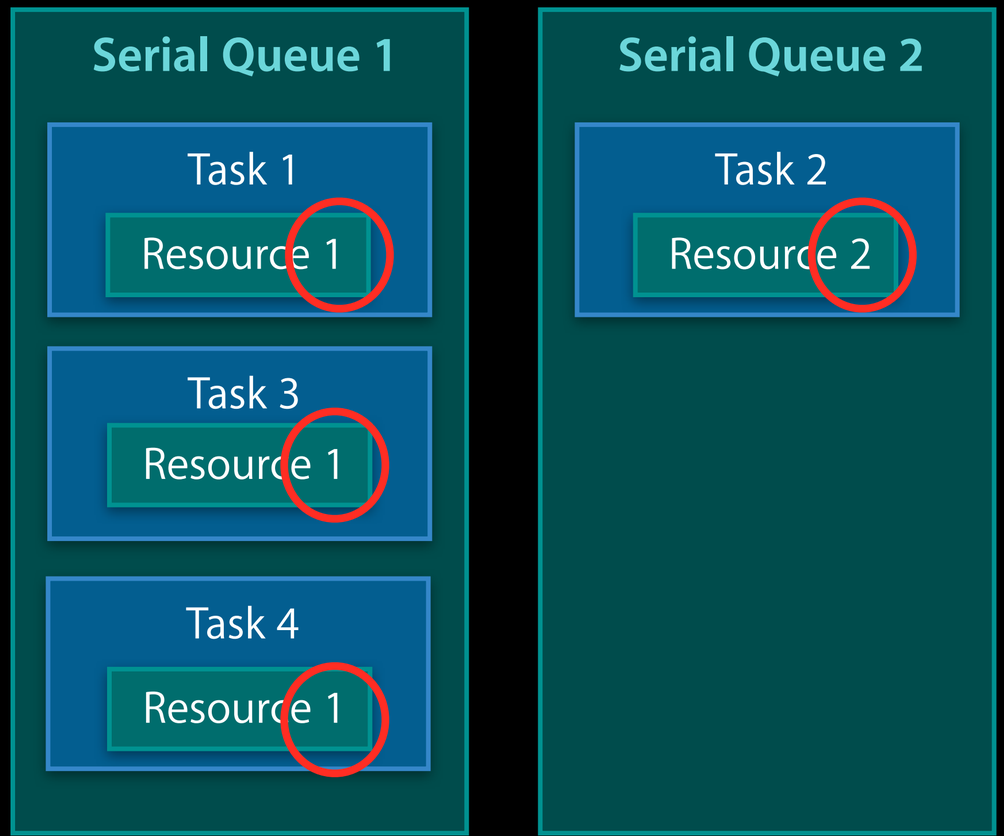
Example: Web

- Clients and servers own their resources
- Communication by http
- Complexity encapsulated

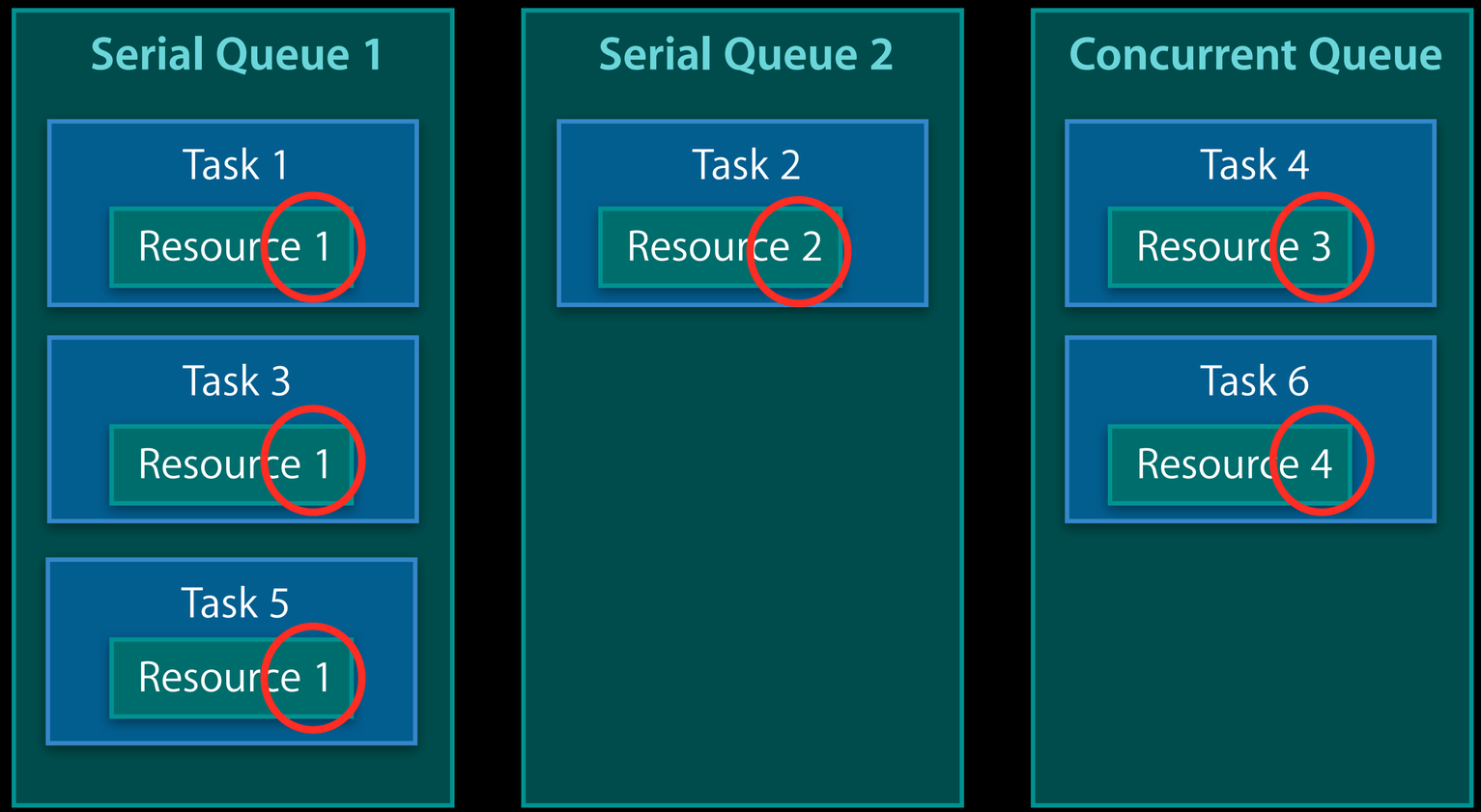
Mechanisms for Isolation

- Dispatch queues default serialized
- NSOperationQueue set maximum concurrency to one
- Thread local variables
- Context pointer parameter to work functions

Isolation on Queues



Isolation with Concurrent Queues



Isolation

- API boundary between tasks
- Resources confined like private ivars
- Isolated code avoids locking overhead

Message Passing

- Formal communication channel
- Avoids exploiting shared memory
- Enqueues request instead
  - NSRunLoop
  - http request response cycle
- Receiver controls message queue

Cocoa Main Thread Support

- NSOperationQueue
  - mainQueue
- Grand Central Dispatch
  - dispatch_get_main_queue
- NSObject
  - performSelectorOnMainThread:withObject:waitUntilDone:
- NSRunLoop
  - mainRunLoop

Coordination

- Prune complexity
- Manage resources safely
- Locks
- Isolation

Coordination Cost

- Time spent not doing primary work
- Overhead compared to serial tasks
- Locks and communication channels are resources
- Often introduces new problems

Resource Contention

Resources

What you have to work with

- CPU
- Memory
- I/O
- Network latency
- Locks

CPU Scheduling

Lots of OS support

- Threads
- Grand Central Dispatch and dispatch queues
- NSOperationQueues
- Processes

Scheduling Non-CPU Resources

By hand

- Airline reservation model
- Slightly overbooked
- Avoid overwhelming resource

Scheduling Non-CPU Resources

Alternatively, by proxy

- Isolation design pattern
- Pair resource with thread or queue
- Leverage OS facilities for CPU scheduling

Resources Constraints

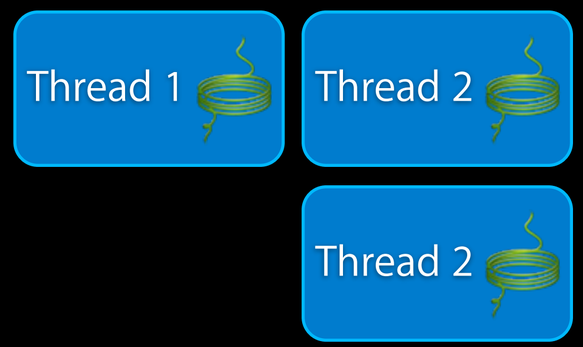
- Total capacity
- Bandwidth
- Latency
- Degradation under load

I/O Efficiency

- Sequential I/O faster
- Fewer, larger requests much faster
- Seek time 10–20ms
- Laptop drives much slower

I/O Contention

- Serial requests can outperform interleaved requests



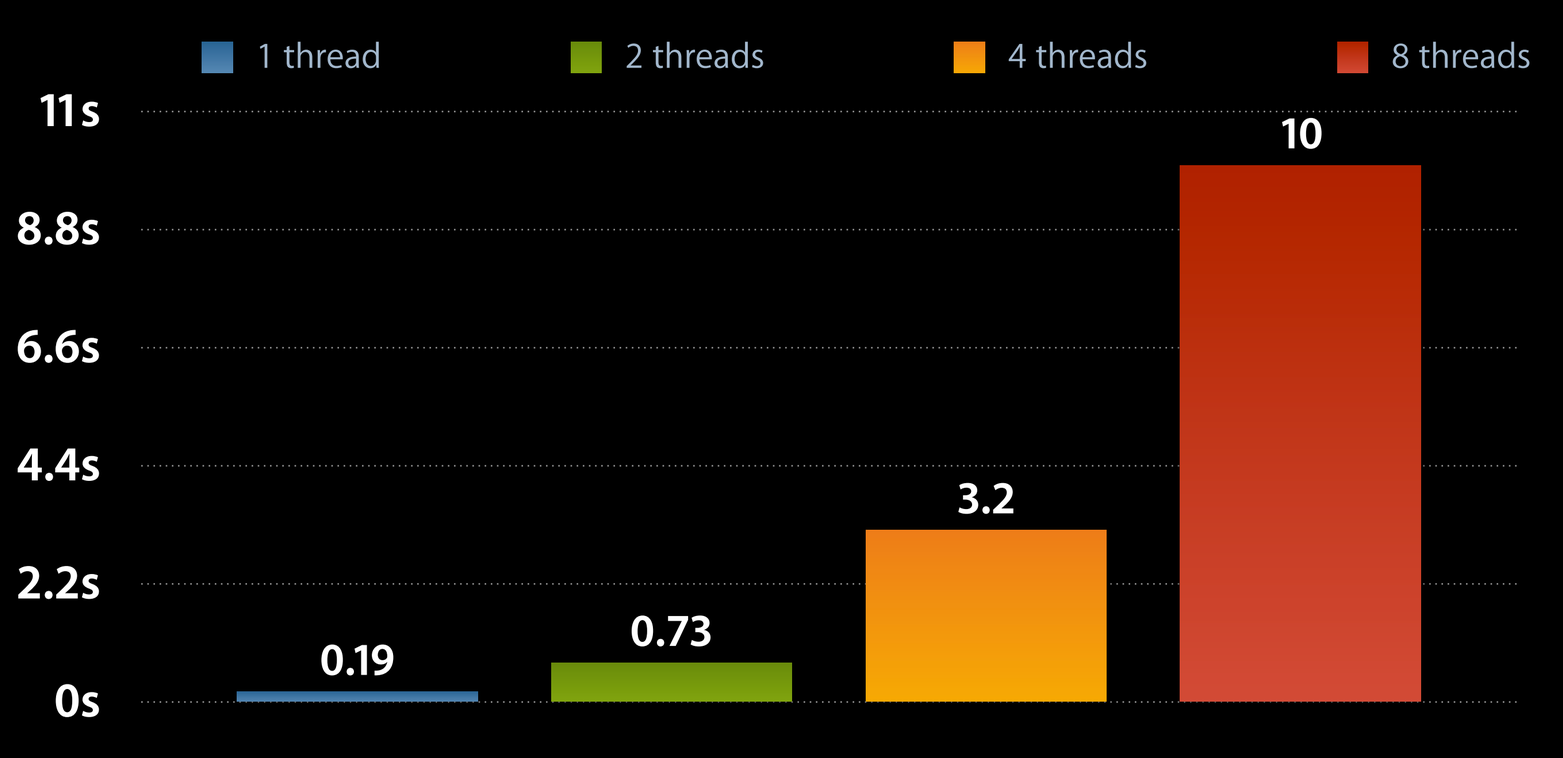
40% Faster

Multicore, Not Multidisk

- Eight cores is not eight hard drives
- I/O is expensive task
- Many small random access I/Os bad
- Find ideal granularity for I/O

Lock Contention

- One very fast lock
- Degrades rapidly with more cores
- Eight thread contention 50x slower



Lock Contention

- Locks without contention fast
- Contended locks very slow
- Contention issues involve hardware
- Atomic operations degrade similarly

Task Representation

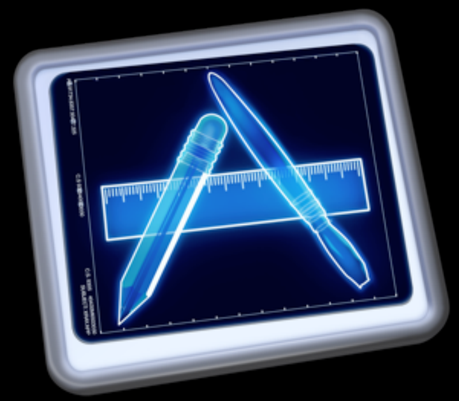
- Pending tasks use memory
  - Blocks, queues cheap not free
  - Threads expensive
- Creating thousands upfront inefficient

Producer Consumer Problem

- Tasks consumed by execution
- Overwhelming consumer queue wasteful
- Executed tasks can enqueue new subtasks
  - Recursive Decomposition
- Futures can represent collection of new tasks

Complex Mix of Resources

- Identify bottlenecks
- Will more threads help or hurt?
- Measure, don’t guess



Resource Contention

- Thread safety != Concurrency
- Some resources degrade badly
  - I/O, Locks
- Avoid overwhelming resources
- Airline reservation model

Task Granularity

Task Granularity

- Balance
  - Available resources
  - Coordination costs
  - Correctness
- Efficient task granularity surprisingly large

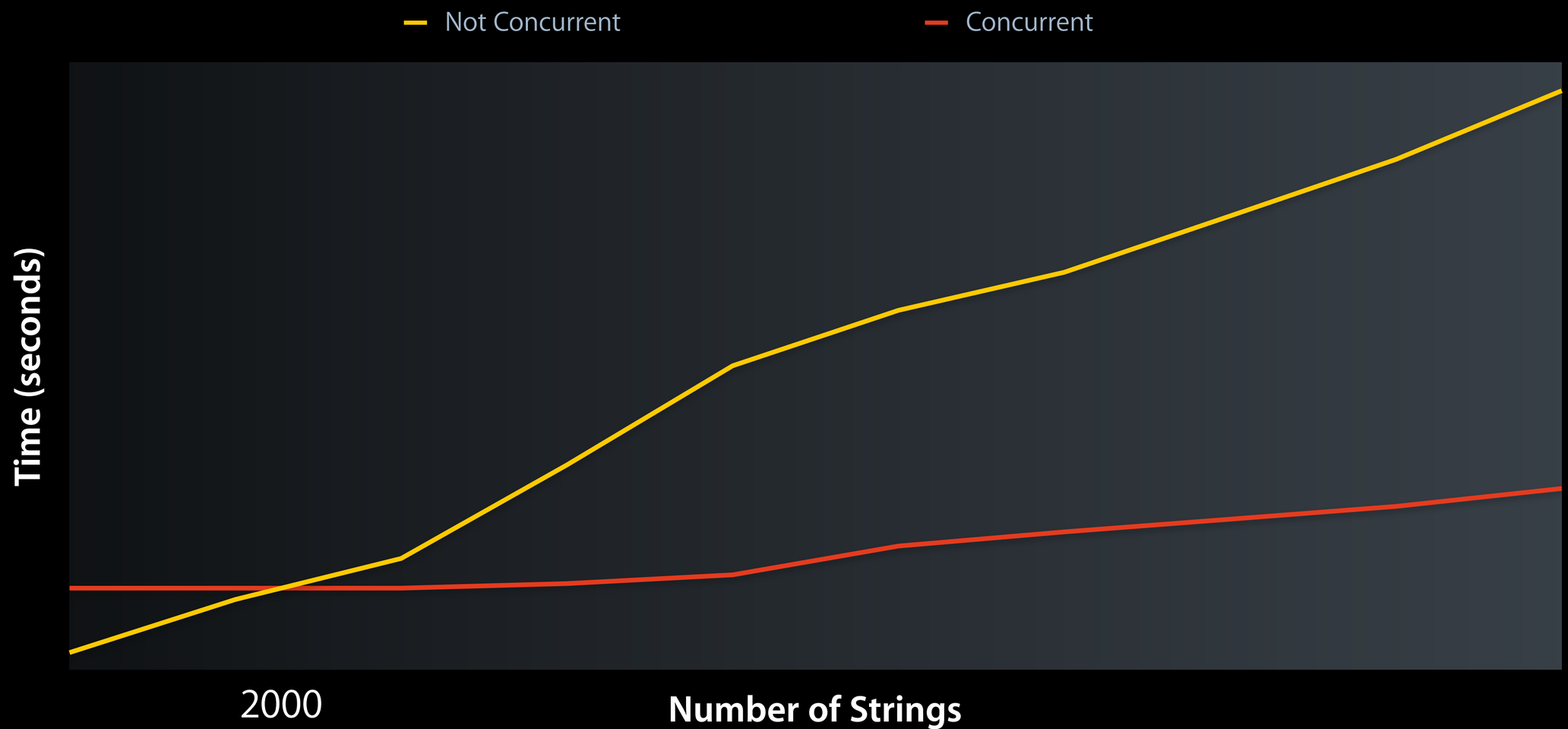
Single Threaded Can Be Faster

- No locking
- No message passing
- No merging results
- No memory to represent tasks
- No scheduling latency

Serialized Better Understood

- Older, better researched
- Optimal serialized algorithms widely available

Concurrent String Sorting



Maintain Correctness

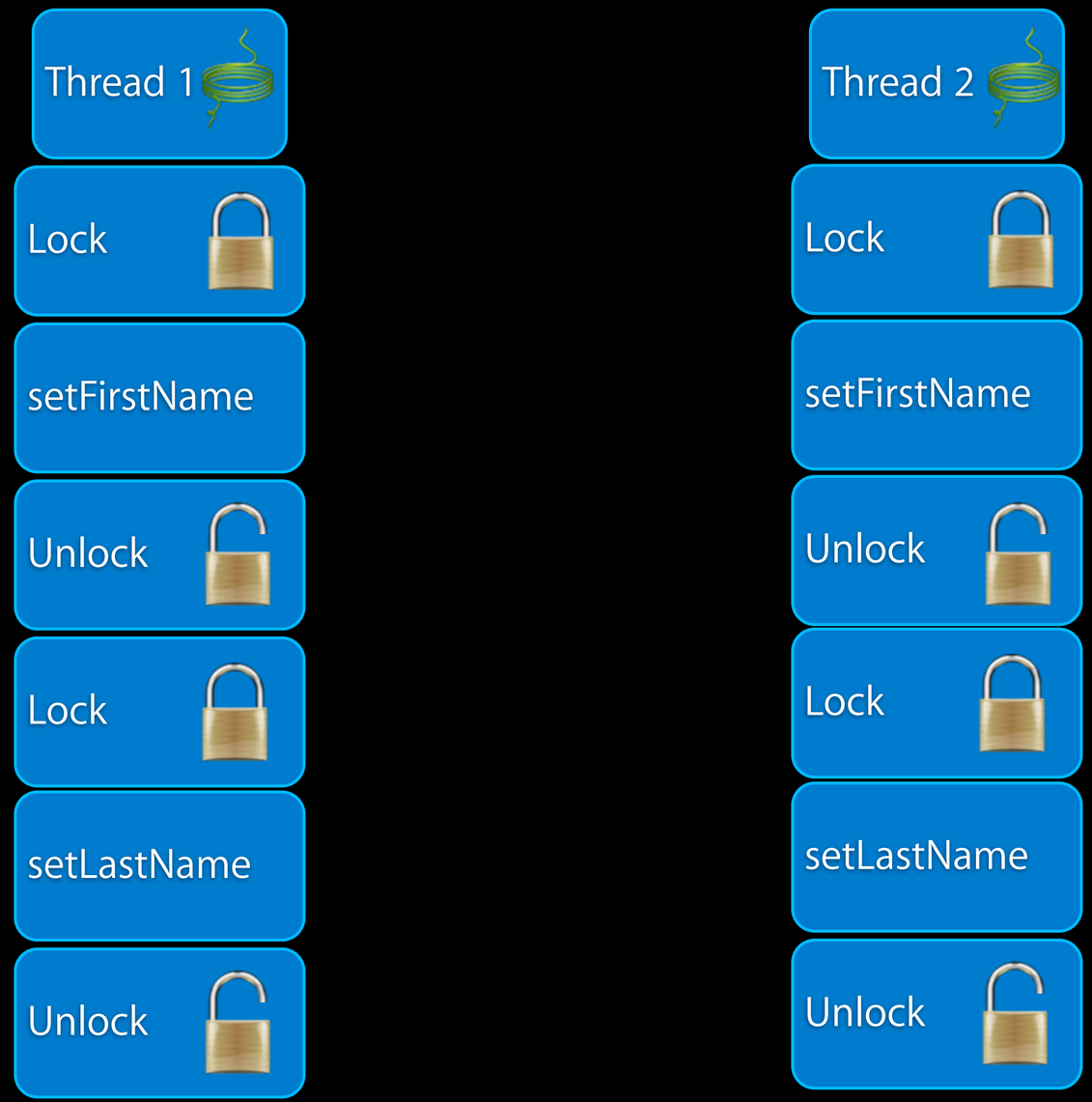
- Scope may have implicit data dependencies
- Avoid subdividing past our assumptions
- Side effects can break preconditions

Interdependent Properties

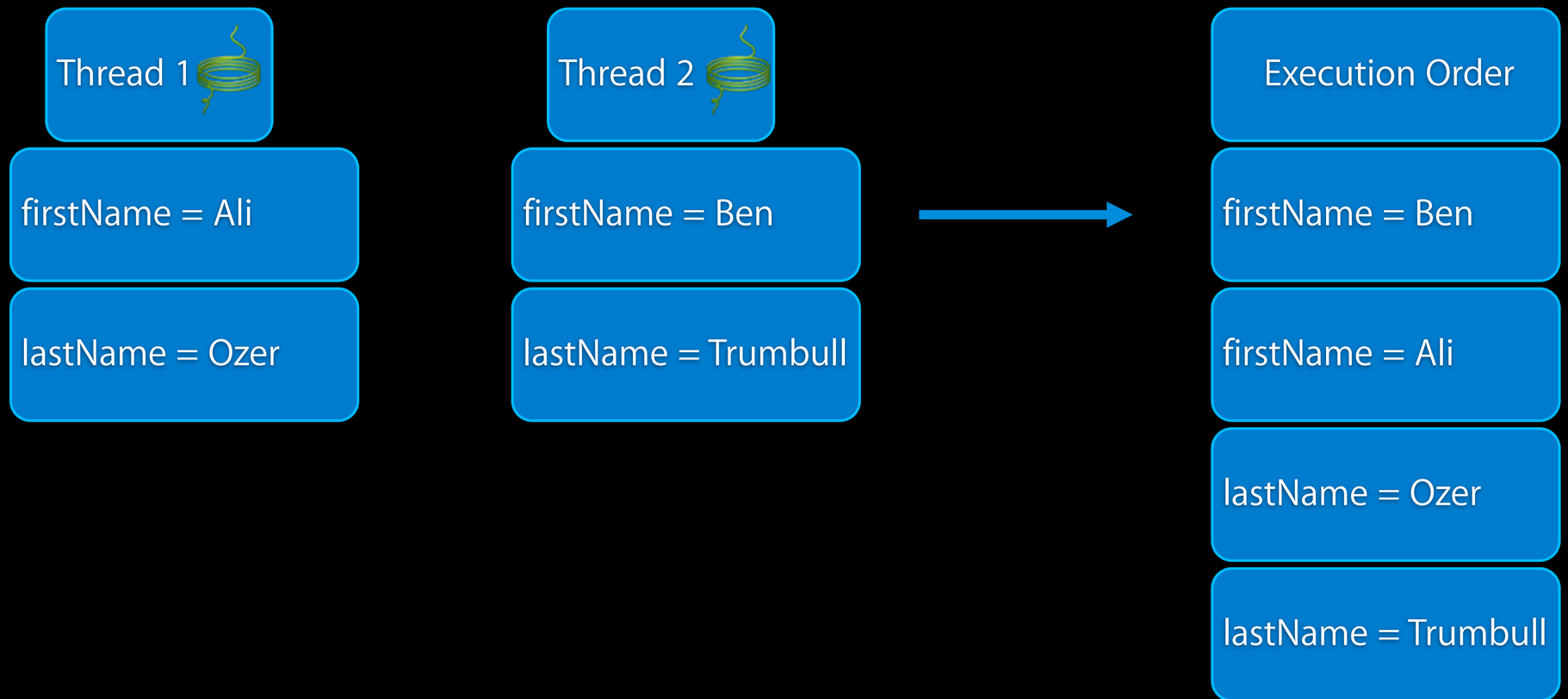
- Objects assume interdependent properties in sync
- Truly independent properties encapsulated as two objects

Example: Names

- Person first and last name



Person Names Independently



Who is Ali Trumbull?

Coarse Grain Tasks

- Less contention over resources
- Amortize coordination cost
- Fewer inter-dependencies between tasks
- Easier to debug
- More difficult to load balance

Fine Grain Tasks

- More scheduling opportunities
- Better load balancing
- Better responsiveness
- More expensive coordination cost

Lots of Knobs

- Focus on primary bottlenecks
- Manage complexity and performance
- Reserve advanced techniques for crucial problems
- Measure, don’t guess

Tuning Granularity

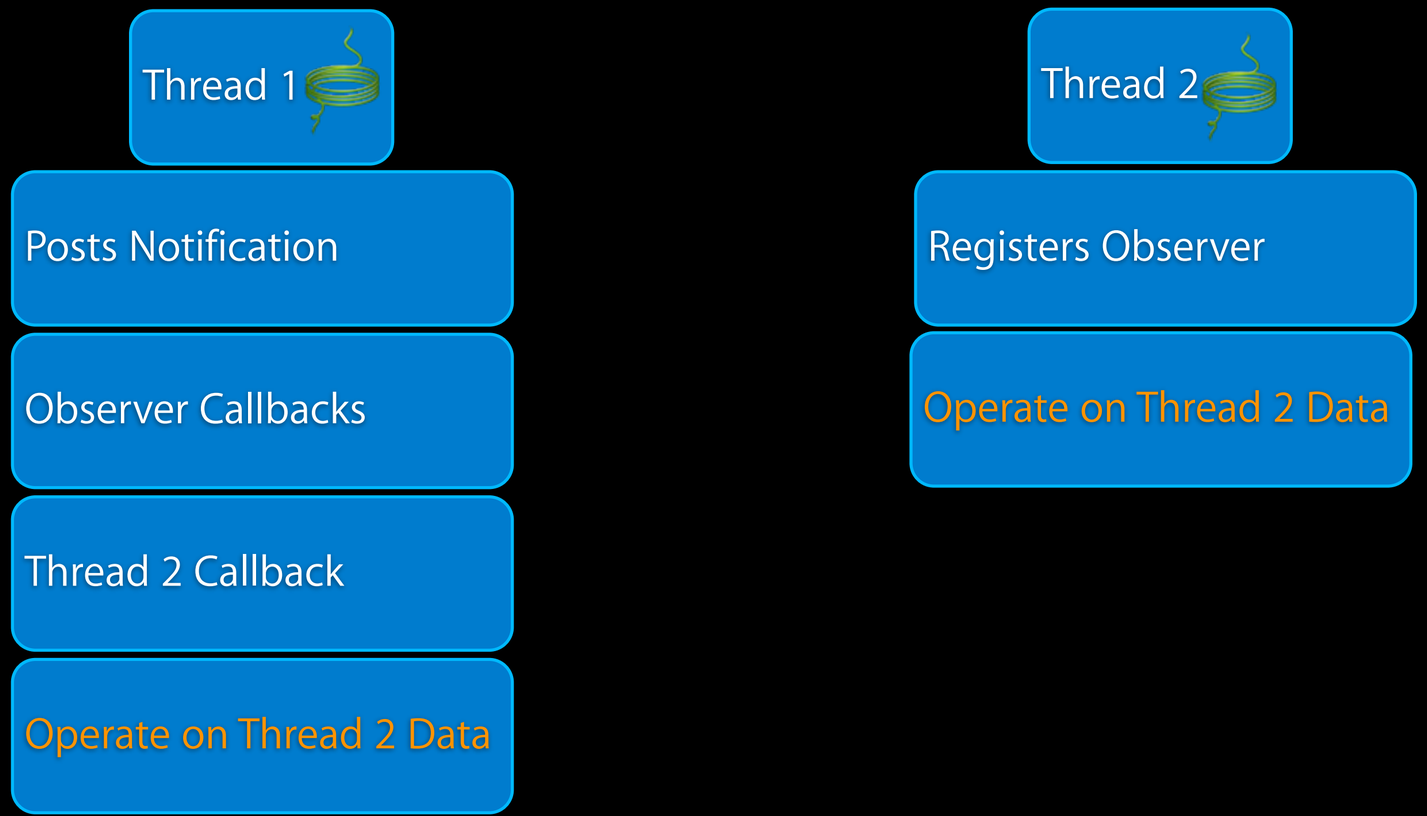
- Break even threshold
- Available resources
- Coordination costs
- Correctness

Multithreaded Notifications

Multithreaded Notifications

- Most notifications delivered synchronously
  - NSNotification
  - Key Value Observing
- Observer actions execute in observee thread
- Visibility in observer thread is completely random

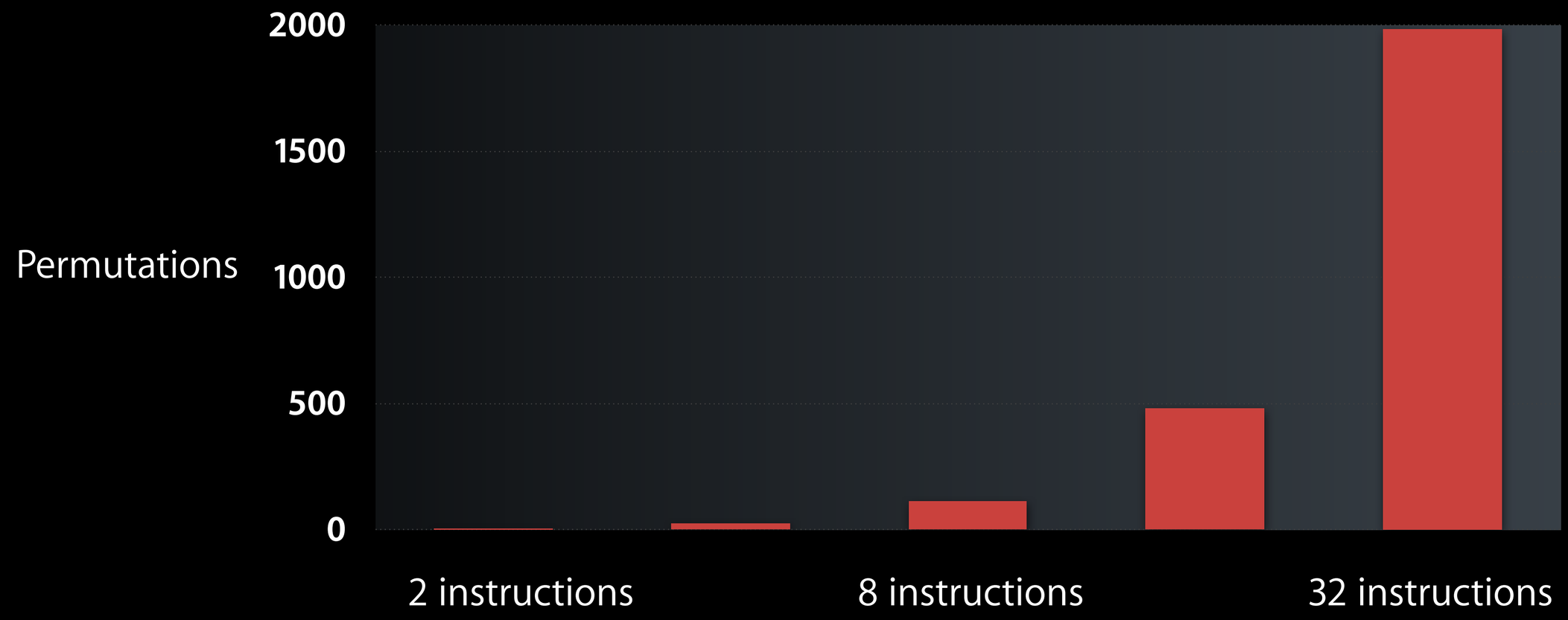
Synchronous Notification Delivery



Multithreaded Notifications Complex

- Shared memory approach perilous
- Introduces many dependencies between threads
- Behavior non-deterministic
- Breaks isolation

2 Threads interleaving



Multithreaded Notifications by Locking

- Ends badly
- API boundaries prevent lock ordering
  - Notification poster ignorant of observee locks
- Prone to deadlock
  - Observees often post new notification during initial callback

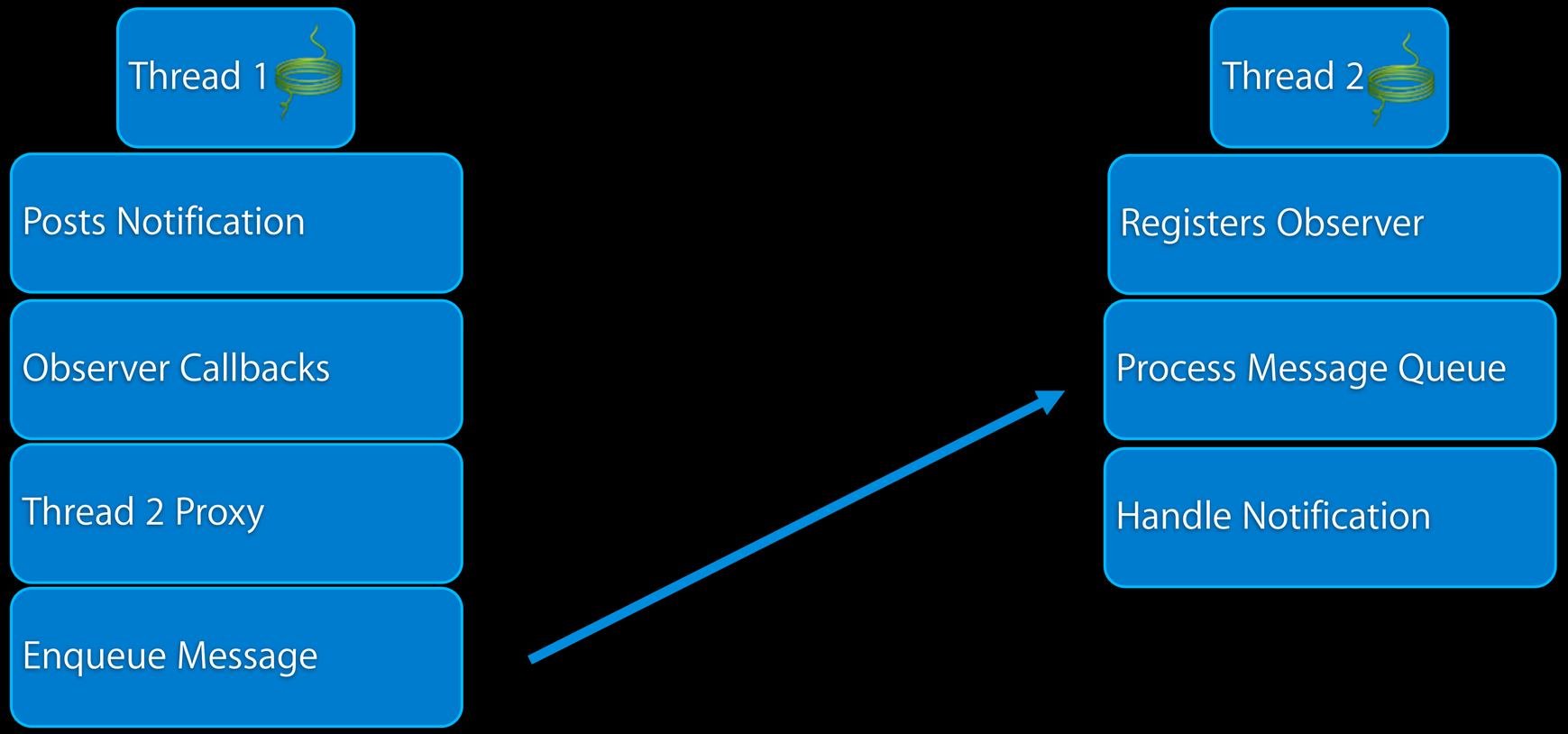
Multithreaded Notifications Without Locking

- Combine Isolation and Message Passing
- Receptionist Pattern

Receptionist Pattern

- Observer is indirect
- Observer proxy and observee isolated in a thread
- Observer proxy and observer use message passing

Receptionist Pattern



Receptionist

- Initial notification isolated in observee thread
- Observer proxy enqueues notification data
- Observer thread processes message queue
- Observers never act via shared memory

Receptionist by NSNotificationCenter

```
- (id)addObserverForName:(NSString *)name
      object:(id)obj
      queue:(NSOperationQueue *)queue
      usingBlock:(void (^)(NSNotification *))block
```

- Block executes in the context of the given queue
- Observer blocks are executed concurrently when possible
  - Posting still synchronous
- If queue is nil, executes in the posting thread

Queues as Receptionists

Cocoa Tips and Tricks	Nob Hill Thursday 5:00PM

Other Receptionist Tasks

- Filter and coalesce notifications
- Re-prioritize notifications
- Discard stale notifications

Passing Data Between Threads

- Never know when receiver will run
- Maybe completes entirely before sender does anything
  - And releases object sender was still using
- Maybe takes page fault, starts seconds later
  - After sender releases object

Keep Objects Alive

- Sender retains
- Receiver releases
- Sender retains once for each of multiple receivers
- Ownership of a retain transfers from sender to receiver

Key Value Observing

- Often many tiny notifications
- Challenging to fit with minimum useful granularity
- High coordination cost from sheer volume

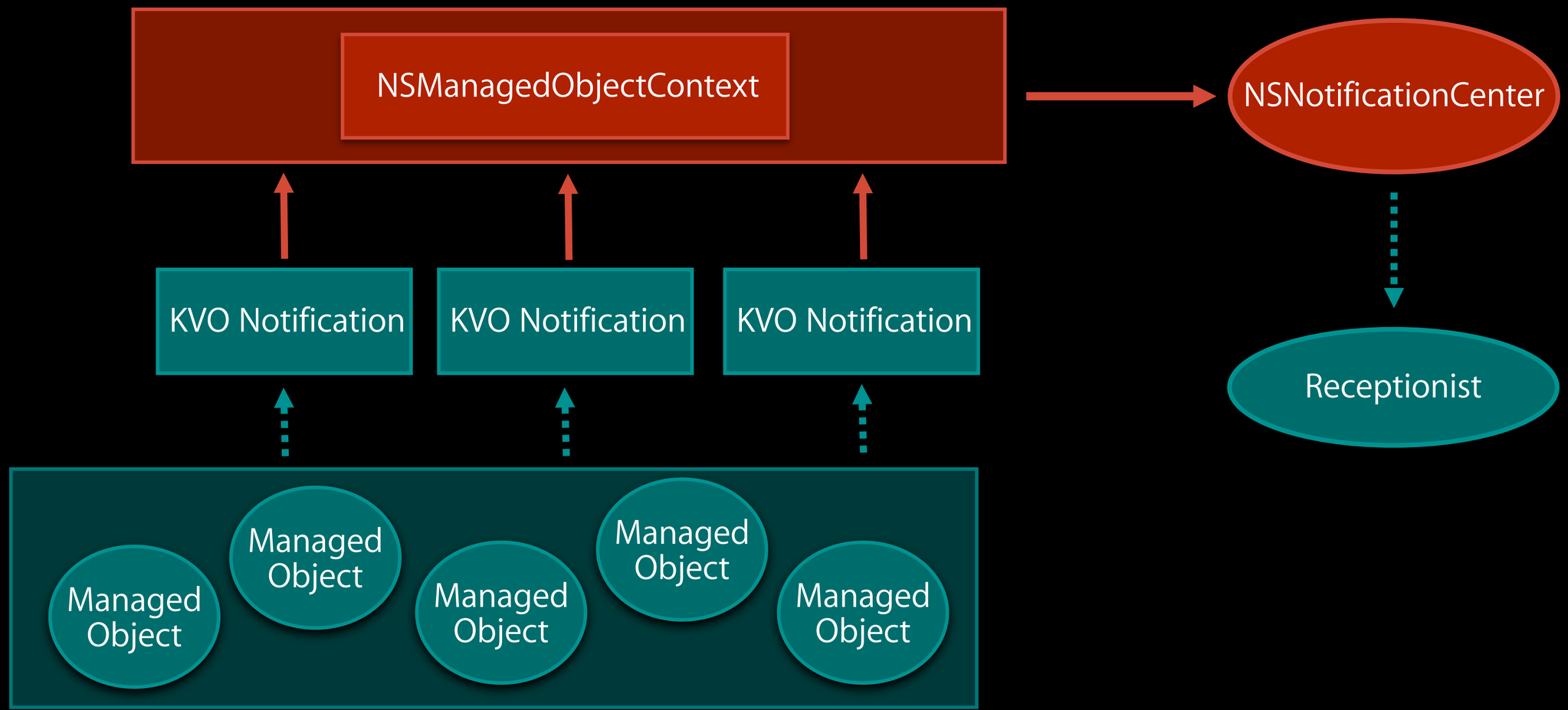
KVO and Isolation

- KVO great with isolation
- Observers and observees confined to a thread
- Amortize inter-thread coordination costs

Multithreading with KVO

- Best coalesced within a thread first
- Consider repackaging in a batch
- NSNotification often fits inter-thread granularity better

KVO Example: Core Data



Coalescing KVO Notifications

- Each context's model objects confined to one thread
- Properties observable within thread
- Context accumulates changes until end of event
- Context reposts coarser grained NSNotification

Multithreaded Notifications

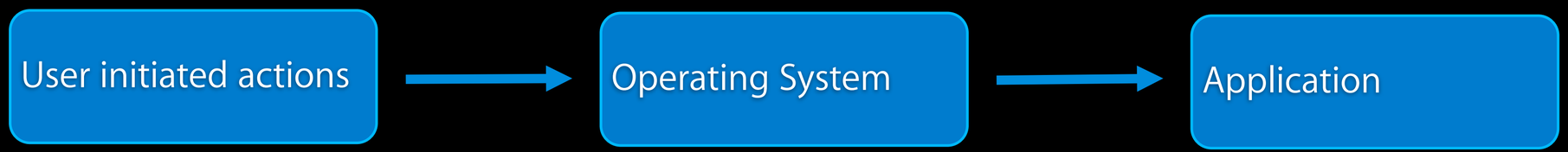
- Avoid locking
- Use message passing
- Receptionist pattern
- Queues make good receptionists
- Coalesce small, frequent notifications

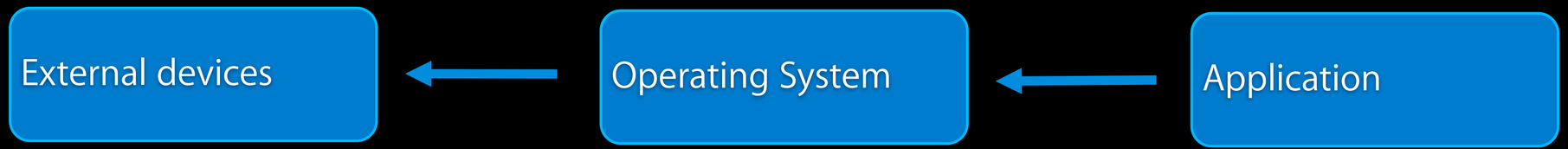
UI and Multithreading

UIs Single Threaded

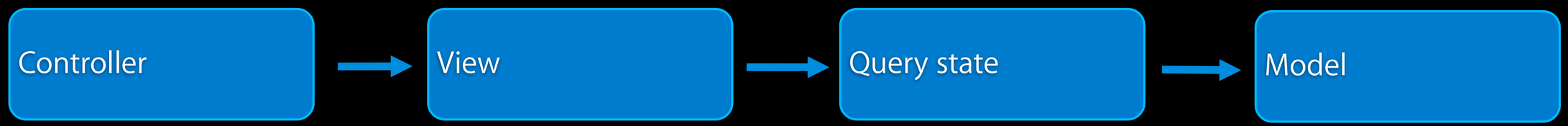
- Multithreaded UIs prone to deadlock
- Inconsistent lock ordering
  - Actions' control flow
  - MVC design pattern

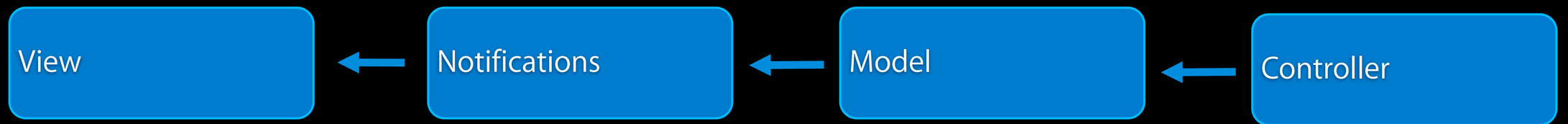
Actions' Control Flow Bidirectional





MVC Design Pattern





UI and Multithreading

- UIs prefer thread confinement
- Bound to main thread
- Views and Controllers

UI Bound to Main Thread

- Doesn’t know about your locks
- Locks require all to participate in protocol
- Locking model in UI cannot be made safe

UI Can Dispatch Operations

- Move blocking operations off main thread
- I/O intensive operations
- Network requests
- Large, expensive operations

Merge Results Cooperatively

- Merge results back into UI
- Message passing
  - Dispatch onto the main queue
  - Perform selector on main thread

Example: NSOperationQueue

```
- (IBAction)doLongTask:(NSButton*)send {
    [myAppQueue addOperationWithBlock:^{
        // long task
        [[NSOperationQueue mainQueue] addOperationWithBlock:^{
            // merge results into UI on main thread
        }];
    }];
}
```

Example: Grand Central Dispatch

```
- (IBAction)doLongTask:(NSButton*)send {
    dispatch_async(dispatch_get_global_queue(0,0), ^{
        // long task
        dispatch_async(dispatch_get_main_queue(), ^{
            // merge results into UI on main thread
        });
    });
}
```

Summary

Summary

- Decompose tasks above break even threshold
- Consider resource limitations
- Avoid excessive coordination
- Prune complexity

Reading Material

- Mac OS X “Threading Programming Guide”
  - http://developer.apple.com/documentation/Cocoa/Conceptual/Multithreading/Introduction/Introduction.html
- “Real World Concurrency”
  - http://portal.acm.org/citation.cfm?doid2=1454456.1454462
- “The Problem with Threads”
  - http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-1.pdf
- “Java Concurrency in Practice” Goetz, et al.
- “Pattern Language for Parallel Programming” Mattson et al.

Related Sessions

Concurrent Programming in Cocoa	North Beach Wednesday 9:00AM
Programming with Blocks and Grand Central Dispatch	Marina Wednesday 2:00PM
Understanding Grand Central Dispatch in Depth	Presidio Thursday 9:00AM
Migrating Your Application to Grand Central Dispatch	Pacific Heights Thursday 2:00PM
Cocoa Tips and Tricks	Nob Hill Thursday 5:00PM

Labs

Grand Central Dispatch Lab	Snow Leopard Lab Wednesday 3:00PM
Concurrent Programming in Cocoa Lab	Lab D Thursday 9:00 AM
Cocoa Lab	Lab C Thursday 2:00PM
Grand Central Dispatch Lab	Snow Leopard Lab Thursday 3:00PM

More Information

Michael Jurewitz
Application Technologies Evangelist
jurewitz@apple.com

Documentation
Mac OS X Threading Programming Guide
http://developer.apple.com/documentation/cocoa/conceptual/multithreading

WWDC Session Survey

You can rate your sessions on the WWDC Attendee Site.

Survey forms are located on each session description page at:

developer.apple.com/wwdc/attendee

WWDC09